multiPoint

MDO Lab

Feb 21, 2022

CONTENTS

`multiPointSparse` is a helper module for doing multipoint optimizations. The documentation `multiPoint` is given below.

To install, first clone the repository, then go into the root directory and type:

```
pip install .
```

For stability we recommend cloning or checking out a tagged release.

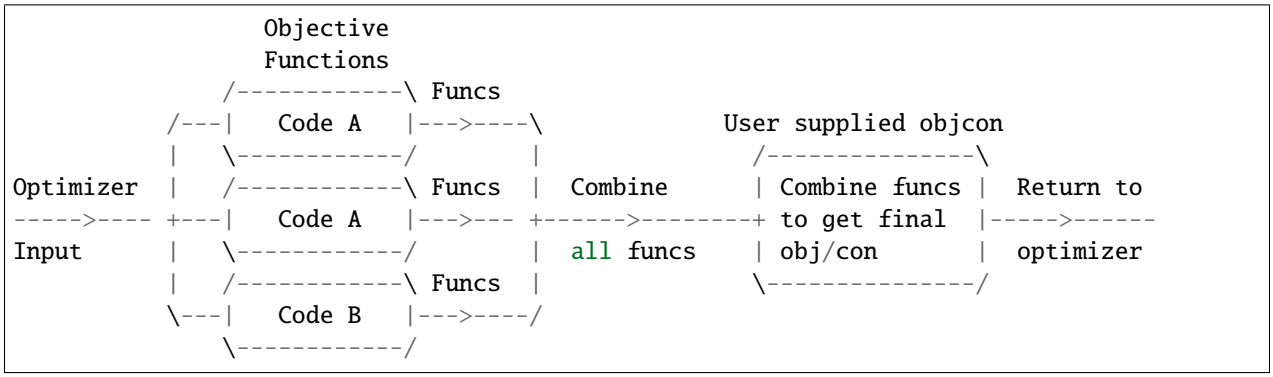
CHAPTER

ONE

TUTORIAL

The goal of `MultiPoint` is to facilitate optimization problems that contain may different computations, all occurring in parallel, each of which may be parallel. `MultiPoint` effectively hides the required MPI communication from the user which results in more readable, more robust and easier to understand optimization scripts.

For our simple example, lets assume we have two parallel codes, `A` and `B` that we want to run at the same time for an optimization. The computations of `A` and `B` do not directly depend on each other; that is they can be executed in an embarrassingly parallel fashion. Lets say we need to run 2 copies of code `A`, and one copy of code `B`. The analysis path would look like:



Lets also assume, that the first copy of code `A` requires 3 processors, the second copy of code `A` requires 2 processors and the copy of code `B` requires 4 processors. For this case, we would require `3 + 2 + 4 = 9` total processors. Scripts using `MultiPointSparse` must be called with precisely the correct number of processors.

```
>>> from mpi4py import MPI
>>> from multipoint import multiPointSparse
>>> MP = multiPointSparse(MPI.COMM_WORLD)
>>> MP.addProcessorSet('codeA', 2, [3, 2])
>>> MP.addProcessorSet('codeB', 1, 4)
>>> comm, setComm, setFlags, groupFlags, ptID = MP.createCommunicators()
>>> setName = MP.getSetName()
```

At this point, you should have the following if you executed the code with 9 processors:

```
>>> print("setName={}, comm.rank={}, comm.size={}, setComm.rank={}, setComm.size={},␣
↪setFlags={}, ptID={}".format(setName, comm.rank, comm.size, setComm.rank, setComm.size,
↪ setFlags, ptID))
setName=codeA, comm.rank=0, comm.size=3, setComm.rank=0, setComm.size=5, setFlags={'codeA
↪': True, 'codeB': False}, ptID=0
setName=codeA, comm.rank=1, comm.size=3, setComm.rank=1, setComm.size=5, setFlags={'codeA
↪': True, 'codeB': False}, ptID=0
```

(continues on next page)

(continued from previous page)

```
setName=codeA, comm.rank=2, comm.size=3, setComm.rank=2, setComm.size=5, setFlags={'codeA
↪': True, 'codeB': False}, ptID=0
setName=codeA, comm.rank=0, comm.size=2, setComm.rank=3, setComm.size=5, setFlags={'codeA
↪': True, 'codeB': False}, ptID=1
setName=codeA, comm.rank=1, comm.size=2, setComm.rank=4, setComm.size=5, setFlags={'codeA
↪': True, 'codeB': False}, ptID=1
setName=codeB, comm.rank=0, comm.size=4, setComm.rank=0, setComm.size=4, setFlags={'codeA
↪': False, 'codeB': True}, ptID=0
setName=codeB, comm.rank=1, comm.size=4, setComm.rank=1, setComm.size=4, setFlags={'codeA
↪': False, 'codeB': True}, ptID=0
setName=codeB, comm.rank=2, comm.size=4, setComm.rank=2, setComm.size=4, setFlags={'codeA
↪': False, 'codeB': True}, ptID=0
setName=codeB, comm.rank=3, comm.size=4, setComm.rank=3, setComm.size=4, setFlags={'codeA
↪': False, 'codeB': True}, ptID=0
```

The input to each of the Objective Functions is the (unmodified) dictionary of optimization variables from pyOptSparse. Each code is then required to use the optimization variables as it requires.

The output from each of Objective functions `funcs` is a Python dictionary of computed values. **For computed values that are different for each member in a processorSet or between processorSets it is necessary to use unique keys**. It is therefore necessary for the user to use an appropriate name mangling scheme.

In the example above we have two copies of Code A. In typical usage, these two instances will produce the same *number* and *type* of quantities but at different operating conditions or other similar variation. Since we need these quantities for either the optimization objective or constraints, these values must be given a unique name.

A simple name-mangling scheme is to simply use the `ptID` variable that is returned from the call to *createCommunicators*:

```
def objA(x):
    funcs['A_%d'%ptID] = function_of_x()

    return funcs
```

A similar thing can be done for `B`:

```
def objB(x):
    funcs['B_%d'%ptID] = function_of_x()

    return funcs
```

A `processorSet` is characterized by a single "objective" and "sensitivity" function. For each `processorSet` we must supply Python functions for the objective and sensitivity evaluation.

```
>>> MP.setProcSetObjFunc('codeA', objA)
>>> MP.setProcSetObjFunc('codeB', objB)
>>> MP.setProcSetSensFunc('codeA', sensA)
>>> MP.setProcSetSensFunc('cdoeB', sensB)
```

The functions `sensA` and `sensB` must compute derivatives of the functionals with respect to the design variables defined in the `optProb` Optimization problem class. Derivatives use the dictionary sensitivity return format described in `pyOptSparse` documentation.

`multiPointSparse` will then automatically communicate the values and call the user supplied `objcon` function with the total set of functions. The purpose of `objcon` is to combine functions from the individual objective functions to

form the final objective and constraint dictionary for `pyOptSparse`. A schematic of this process is given below:

```
                    Pass-through keys
                /-----------------------------\
  all funcs     |                               |   output to pyOptSparse
------->------ + input      /--------\ output   |-------------->----
                \-------->--+ objcon | ----------/
                  keys      \--------/ keys
```

`multiPointSparse` analyzes the optimization object and determine if any of the required constraint keys are already present in all funcs, these keys are flagged as "pass-through"…that is they "by-pass" entirely the `objcon` function. The purpose therefore of objcon is to use the remaining functions in `all funcs` (the `input keys`) to compute the remainder of the required constraints (`output keys`) and objective. For example:

```
def objcon(funcs):
    fobj = 0.0
    for i in range(2):
        fobj += funcs['A_%d'%i]

    fobj /= funcs[B_0]
    fcon['B_con'] = funcs[B_0]/funcs[A_0]
    return fobj, fcon
```

There all three values contribute to the objective, while `A_0` and `B_0` combine to form the constraint `B_con`. This example has no `pass-though keys`.

Generally speaking, the computations in objcon should be simple and not overly computationally intensive. The sensitivity of the `output keys` with respect to the `input keys` is computed automatically by `multiPointSparse` using the complex step method.

Warning: Pass-through keys **cannot** be used in objcon.

Warning: Computations in objcon must be able to use complex number. Generally this will mean if numpy arrays are used, the `dtype=complex` keyword argument is used.

The `objcon` function is set using the call:

```
>>> MP.setObjCon(objCon)
```

As noted earlier, `multiPointSparse` uses the optimization problem to determine which keys are already constraints and which need to be combined in `objcon`. This is done using:

```
>>> optProb = Optimization('opt', MP.obj)
>>> # Setup optimization problem
>>> # MP needs the optProb after everything is setup.
>>> MP.setOptProb(optProb)
>>> # Create optimizer and use MP.sens for the sensitivity function on opt call
>>> snopt(optProb, sens=MP.sens, ...)
```

CHAPTER

TWO

EXAMPLES

2.1 Example 1

In this example we show basic operations how to create and access specific sets or processor groups. To start we begin by importing the necessary modules.

```
# ==========================================================================
# Import modules
# ==========================================================================
from mpi4py import MPI
from multipoint import multiPointSparse
```

In this example we assume we have two codes or processor sets, `codeA` and `codeB`. We assume 3 instances of `codeA` where each instance requires 3, 2, and 1 processors. For set `codeB` we assume only 1 instance that requires 4 processors. Following the creation of the points sets the communicators are created.

```
# ==========================================================================
# Processor allocation
# ==========================================================================
# Instantiate the multipoint object
MP = multiPointSparse(MPI.COMM_WORLD)

# Add all processor sets and create the communicators
MP.addProcessorSet("codeA", 3, [3, 2, 1])
MP.addProcessorSet("codeB", 1, 4)
comm, setComm, setFlags, groupFlags, ptID = MP.createCommunicators()

# Extract setName on a given processor for convenience
setName = MP.getSetName()

# Create all directories for all groups in all sets
ptDirs = MP.createDirectories("./output")

# For information, print out all values on all processors
print(
    f"setName={setName}, comm.rank={comm.rank}, comm.size={comm.size}, setComm.rank=
↪{setComm.rank}, setComm.size={setComm.size}, setFlags={setFlags}, groupFlags=
↪{groupFlags}, ptID={ptID}"
)
```

Printing all values gives the following:

```
setName=codeA, comm.rank=1, comm.size=3, setComm.rank=1, setComm.size=6, setFlags={'codeA
↪': True, 'codeB': False}, groupFlags=[ True False False], ptID=0
setName=codeA, comm.rank=2, comm.size=3, setComm.rank=2, setComm.size=6, setFlags={'codeA
↪': True, 'codeB': False}, groupFlags=[ True False False], ptID=0
setName=codeA, comm.rank=0, comm.size=2, setComm.rank=3, setComm.size=6, setFlags={'codeA
↪': True, 'codeB': False}, groupFlags=[False  True False], ptID=1
setName=codeA, comm.rank=1, comm.size=2, setComm.rank=4, setComm.size=6, setFlags={'codeA
↪': True, 'codeB': False}, groupFlags=[False  True False], ptID=1
setName=codeA, comm.rank=0, comm.size=1, setComm.rank=5, setComm.size=6, setFlags={'codeA
↪': True, 'codeB': False}, groupFlags=[False False  True], ptID=2
setName=codeB, comm.rank=0, comm.size=4, setComm.rank=0, setComm.size=4, setFlags={'codeA
↪': False, 'codeB': True}, groupFlags=[ True], ptID=0
setName=codeB, comm.rank=1, comm.size=4, setComm.rank=1, setComm.size=4, setFlags={'codeA
↪': False, 'codeB': True}, groupFlags=[ True], ptID=0
setName=codeB, comm.rank=2, comm.size=4, setComm.rank=2, setComm.size=4, setFlags={'codeA
↪': False, 'codeB': True}, groupFlags=[ True], ptID=0
setName=codeB, comm.rank=3, comm.size=4, setComm.rank=3, setComm.size=4, setFlags={'codeA
↪': False, 'codeB': True}, groupFlags=[ True], ptID=0
setName=codeA, comm.rank=0, comm.size=3, setComm.rank=0, setComm.size=6, setFlags={'codeA
↪': True, 'codeB': False}, groupFlags=[ True False False], ptID=0
```

To perform operations on all processors in a set we can use the setFlags. This is convenient if we want to perform the same operation on all processors in a given set. Furthermore, if we want to access only a specific group in a set, the `ptID` can be conveniently used as shown.

```python
# ================================================================================
# Problem setup
# ================================================================================
# To perform operations on all processors in a set we can use the setFlags
if setFlags["codeA"]:  # Alternatively, setName == "codeA" could be used here
    # ...
    # To access a particular group within the set can be done using the ptID
    # Here we access only the processors in the first group
    if 0 == ptID:
        print(f"setName={setName} comm.rank={comm.rank} ptID={ptID}")

    # To access all groups (but still a specific one) we simply loop over the size of␣
↪the set
    for i in range(setComm.size):
        if i == ptID:
            print(f"setName={setName} comm.rank={comm.rank} ptID={ptID} i={i}")

# Similarly, for the other processor set
if setFlags["codeB"]:
    for i in range(setComm.size):
        if i == ptID:
            print(f"setName={setName} comm.rank={comm.rank} ptID={ptID} i={i}")
```

CHAPTER

THREE

API

class multipoint.multiPointSparse.**multiPointSparse**(*gcomm*)

Create the multiPoint class on the provided comm.

Parameters

gcomm [MPI.Intracomm] Global MPI communicator from which all processor groups are created. It is usually MPI_COMM_WORLD but may be another intraCommunicator that has already been created.

Notes

multiPointSparse requires `useGroups=True` (default) when creating the optProb (Optimization instance).

Examples

We will setup a multipoint problem with two procSets: a 'cruise' set with 3 members and 32 procs each, and a maneuver set with two members with 10 and 20 procs respectively. Our script will have to define 5 python functions:

1. Evaluate functions for cruise:

```
def cruiseObj(x):
    funcs = {} # Fill up with functions
    ...
    return funcs
```

2. Evaluate functions for maneuver:

```
def maneuverObj(x):
    funcs = {} # Fill up with functions
    ...
    return funcs
```

3. Evaluate function sensitivity for cruise:

```
def cruiseSens(x, funcs):
    funcSens = {}
    ...
    return funcSens
```

4. Evaluate function sensitivity for cruise:

```
def maneuverSens(x, funcs):
    funcSens = {}
    ...
    return funcSens
```

5. Function to compute addition functions:

```
def objCon(funcs):
    funcs['new_func'] = combination_of_funcs
    ...
    return funcs
```

```
>>> MP = multiPointSparse.multiPoint(MPI.COMM_WORLD)
>>> MP.addProcessorSet('cruise', 3, 32)
>>> MP.addProcessorSet('maneuver', 2, [10, 20])
>>> # Possibly create directories
>>> ptDirs = MP.createDirectories('/home/user/output/')
>>> # Get the communicators and flags
>>> comm, setComm, setFlags, groupFlags, ptID = MP.createCommunicators()
>>> # Setup problems and python functions
>>> ....
>>> MP.setProcSetObjFunc('cruise', cruiseObj)
>>> MP.setProcSetObjFunc('maneuver', maneuverObj)
>>> MP.setProcSetSensFunc('cruise', cruiseSens)
>>> MP.setProcSetSensFunc('maneuver', maneuverSens)
>>> MP.setObjCon(objCon)
>>> # Create optimization problem using MP.obj
>>> optProb = Optimization('opt', MP.obj)
>>> # Setup optimization problem
>>> # MP needs the optProb after everything is setup.
>>> MP.setOptProb(optProb)
>>> # Create optimizer and use MP.sens for the sensitivity function on opt call
>>> snopt(optProb, sens=MP.sens, ...)
```

addConsAsObjConInputs(*cons*)

This function allows functions to be used both as constraints, as well as inputs to the ObjCon, therefore no longer bypassed.

Parameters

cons [string or list of strings] The constraint names the user wants to use as ObjCon inputs

addDVsAsFunctions(*dvs*)

This function allows you to specify a list of design variables to be explicitly used as functions. Essentially, we just copy the values of the DVs directly into keys in 'funcs' and automatically generate an identity Jacobian. This allows the remainder of the objective/sensitivity computations to be proceed as per usual.

Parameters

dvs [string or list of strings] The DV names the user wants to use directly as functions

addProcSetObjFunc(*setName*, *func*)

Add an additional python function handle to compute the functionals

Parameters

setName [str] Name of set we are setting the function for

func [Python function] Python function handle

`addProcSetSensFunc`(*setName*, *func*)

Add an additional python function handle to compute the derivative of the functionals

Parameters

setName [str] Name of set we are setting the function for

func [Python function] Python function handle

`addProcessorSet`(*setName*, *nMembers*, *memberSizes*)

A Processor set is defined as one or more groups of processors that use the same obj() and sens() routines. Members of processor sets typically, but not necessarily, return the same number of functions. In all cases, the function names must be unique.

Parameters

setName [str] Name of process set. Process set names must be unique.

nMembers [int] Number of members in the set.

memberSizes [int, iteratable] Number of processors on each set. If an integer is supplied all members use the same number of processors. If a list or array is provided, a different number of processors on each member can be specified.

Examples

```
>>> MP = multiPointSparse.multiPoint(MPI.COMM_WORLD)
>>> MP.addProcessorSet('cruise', 3, 32)
>>> MP.addProcessorSet('maneuver', 2, [10, 20])
```

The `cruise` set creates 3 processor groups, each of size 32. and the `maneuver` set creates 2 processor groups, of size 10 and 20.

`createCommunicators`()

Create the communicators after all the procSets have been added. All procSets MUST be added before this routine is called.

Returns

comm [MPI.Intracomm] This is the communicator for the member of the procSet. Basically, this is the communicator that the (parallel) analysis should be created on.

setComm [MPI.Intracomm] This is the communicator that spans the entire processor set.

setFlags [dict] This is a dictionary whose entry for `setName`, as specified in addProcessorSet() is True on a processor belonging to that set.

groupFlags [list] This list is used to distinguish between members within a processor set. This list of of length nMembers and the ith entry is true for the ith group.

ptID [int] This is the index of the group that this processor belongs to.

Examples

```
>>> MP = multiPointSparse.multiPoint(MPI.COMM_WORLD)
>>> MP.addProcessorSet('cruise', 3, 32)
>>> MP.addProcessorSet('maneuver', 2, [10, 20])
>>> comm, setComm, setFlags, groupFlags, ptID = MP.createCommunicators()
```

The following will be true for all processors for the second member of the `cruise` procSet.

```
>>> setFlags['cruise'] and groupFlags[1] == True
```

createDirectories(*rootDir*)

This function can be called only after all the procSets have been added. This can facilitate distinguishing output files when there are a large number of procSets and/or members of procSets.

Parameters

rootDir [str] Root path where directories are to be created

Returns

ptDirs [dict] A dictionary of all the created directories. Each dictionary entry has key defined by 'setName' and contains a list of size nMembers, each entry of which is the path to the created directory

Examples

```
>>> MP = multiPointSparse.multiPoint(MPI.COMM_WORLD)
>>> MP.addProcessorSet('cruise', 3, 32)
>>> MP.addProcessorSet('maneuver', 2, [10, 20])
>>> ptDirs = MP.createDirectories('/home/user/output/')
>>> ptDirs
{'cruise': ['/home/user/output/cruise_0',
            '/home/user/output/cruise_1',
            '/home/user/output/cruise_2'],
 'maneuver': ['/home/user/output/maneuver_0',
              '/home/user/output/maneuver_1']}
```

getSetName()

After MP.createCommunicators is call, this routine may be called to return the name of the set that this processor belongs to. This may result in slightly cleaner script code.

Returns

setName [str] The name of the set that this processor belongs to.

obj(*x*)

This is a built-in objective function that is designed to be used directly as an objective function with pyOptSparse. The user should not use this function directly, instead see the class documentation for the intended usage.

Parameters

x [dict] Dictionary of variables returned from pyOptSparse

`sens`(*x*, *funcs*)
This is a built-in sensitivity function that is designed to be used directly as a the sensitivity function with pyOptSparse. The user should not use this function directly, instead see the class documentation for the intended usage.

Parameters

x [dict] Dictionary of variables returned from pyOptSparse

`setObjCon`(*func*)
Set the python function handle to compute the final objective and constraints that are combinations of the functionals.

Parameters

func [Python function] Python function handle

`setOptProb`(*optProb*)
Set the optimization problem that this multiPoint object will be used for. This is required for this class to know how to assemble the gradients. If the optProb is not 'finished', it will done so here. Therefore, this function is collective on the comm that optProb is built on. multiPoint sparse does *not* hold a reference to optProb so no additional changes can be made to optProb after this function is called.

Parameters

optProb [pyOptSparse optimization problem class] The optProb object to use

`setProcSetObjFunc`(*setName*, *func*)
Set a single python function handle to compute the functionals

Parameters

setName [str] Name of set we are setting the function for

func [Python function] Python function handle

`setProcSetSensFunc`(*setName*, *func*)
Set the python function handle to compute the derivative of the functionals

Parameters

setName [str] Name of set we are setting the function for

func [Python function] Python function handle

INDEX